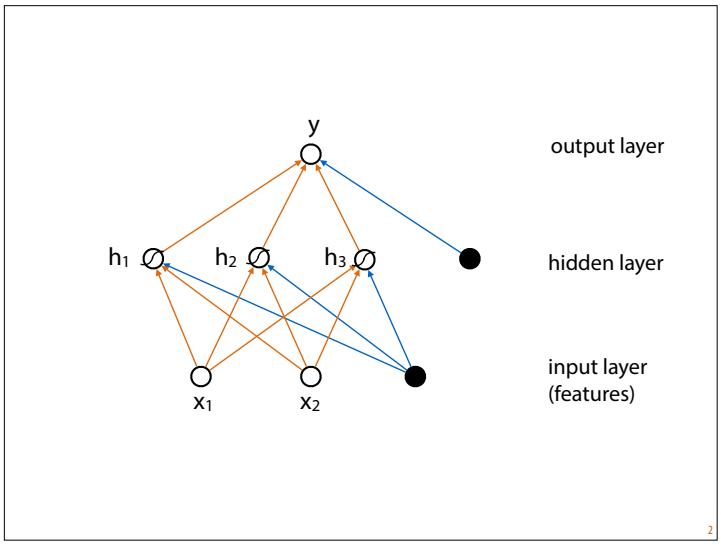
Deep Learning 1

Machine Learning 2019
mlvu.github.io

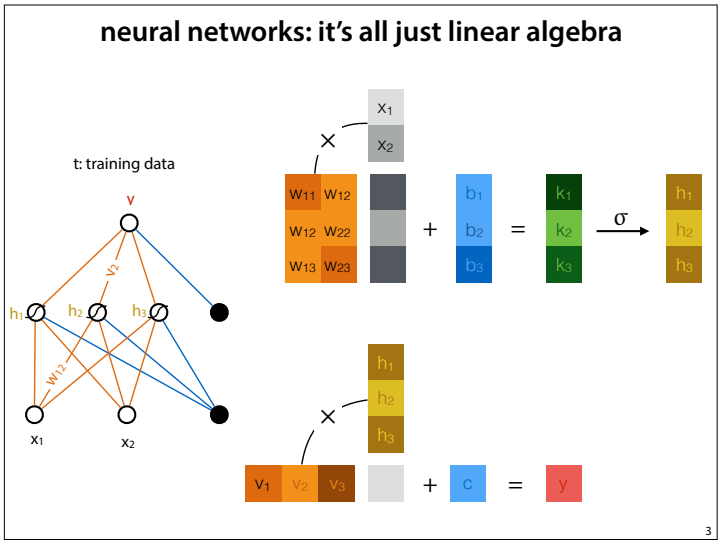


We are picking up the discussion about neural networks from the last lecture. A neural network is model described by a graph: each node represents a scalar value. The value of each node is computed from the incoming edges by multiplying the weight on the edge by the value of the node it connects to.

We train the model by tuning the weights.

The feedforward network is the simplest way of wiring up a neural net, but we will see other possibilities later.

Every orange and blue line in this picture represents one parameter of the model.



Last lecture, we discussed neural networks from the perspective of a graphical model. Dots joined by arrows, to describe a computation.

However, there's another perspective that can greatly simplify things. Most of the operations used in neural networks can also be written as matrix multiplications.

$$f(\mathbf{x}) = \mathbf{V}\,\sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

As you can see, this greatly simplifies our notation. It also allows for very efficient implementation of neural networks, since matrix multiplication can be implemented very efficiently.

This is what we will discuss today: how to simplify the basic idea of neural networks into a very powerful and flexible framework for creating highly complex machine learning models.

---

deep learning

part 1:

Deep learning systems
tensors, matrix calculus, backpropagation revisited

Backpropagation revisited
Multivariate chain rule, calculus over tensors

part 2:

Making it work
vanishing gradients, minibatches, optimizers, regularization

Convolutional neural networks
aka ConvNets or CNNs

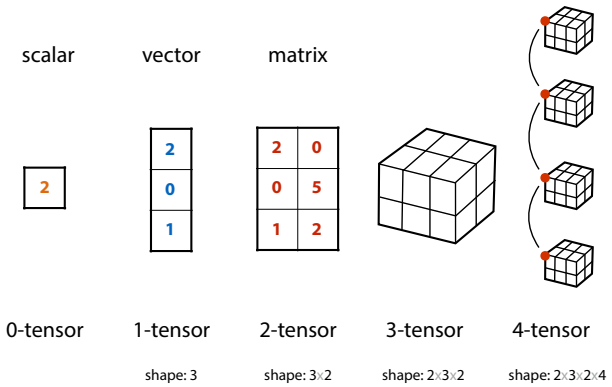
What is deep learning?

5

Autoencoders have been moved to “Deep learning 2”

---

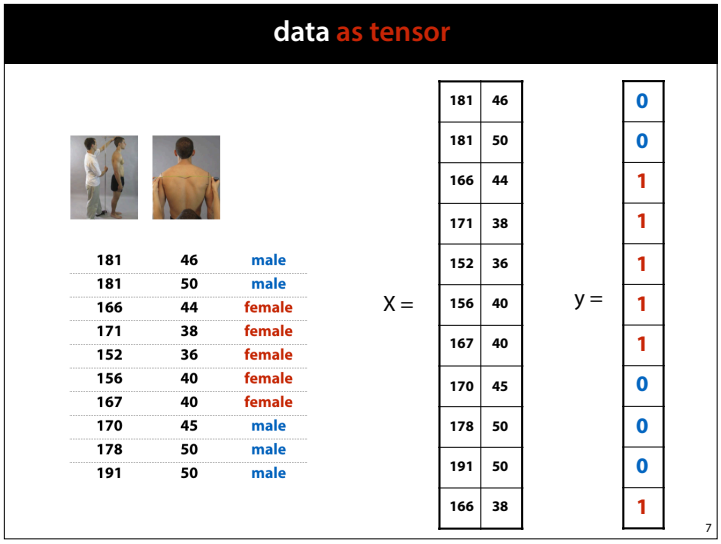
tensors



6

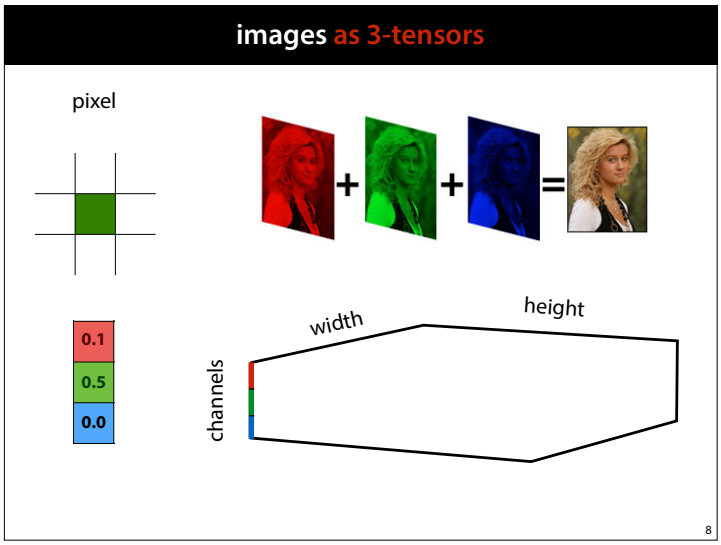
The main ingredient in the language of deep learning is the **tensor**. For our purposes, a tensor is nothing more than a straightforward generalisation of vectors and matrices to higher dimensionalities. We can deal with any kind of data so long as it can be cast as a tensor.

Let’s look at some examples of how data is represented in tensor form.



A simple dataset, with numeric features can simply be represented as a matrix (with a corresponding vector for the labels).

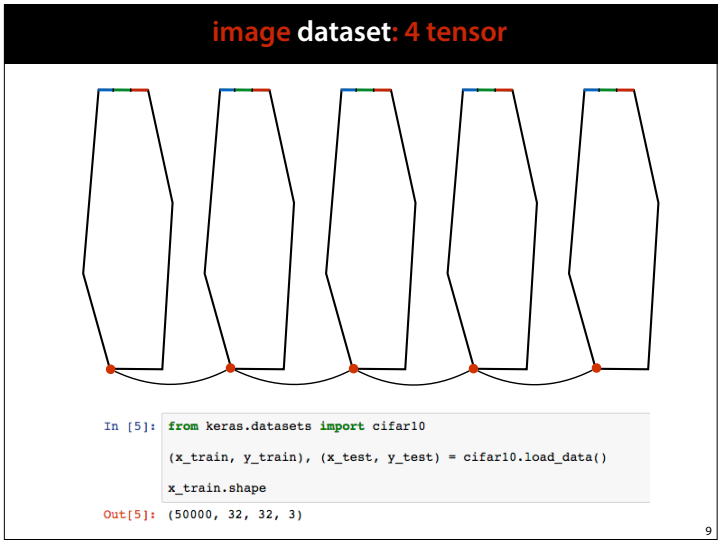
Any categoric features or labels should be converted to numeric features (see the Methodology 2 lecture).



Images can be represented as 3-tensors. In an RGB image, the color of a single pixel is represented using three values between 0 and 1 (how red it is, how green it is and how blue it is). This means that an RGB image can be thought of as a stack of three color channels, represented by matrices.

This stack is a 3-tensor.

source: ***http://www.adsell.com/scanning101.html***

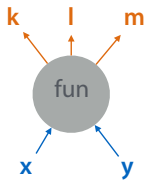


If we have a dataset of multiple images, we get a 4-tensor. Theimage dimensions we saw already, and one extra, indexing the different images in the data.

The Keras code shows how this looks in python: we load the CIFAR 10 dataset. The training data contains 50 000 images, each with 32 by 32 resolution and 3 color channels.

a deep learning system

Consists of *functions*.
Also called operations



Functions can have multiple inputs and outputs. All inputs, outputs are *tensors*.

Functions implement:
- forward(…) computing the outputs given the inputs
- backward(…) computing the local derivative

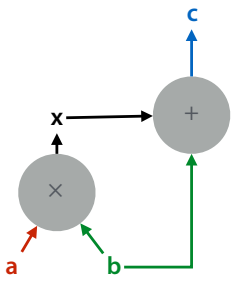
Training: the user chains modules into a *computation graph* that produces a scalar loss. The loss is backpropagated to compute the gradient.

10

This is the basic principle of a deep learning system. We define modules that we chain together into a **computation graph**. The modules implement a local derivative (more on this later) in addition to the forward computation. This allows us to compute the gradient through backpropagation.

---

computation graph for (a × b) + b = c



11

This is what we mean by a computation graph, a directed acyclic graph that shows the data (tensors a, b, c, x) flowing through the functions.

A deep learning system uses a computation graph to execute a given computation (**the forward pass**) and then uses the backpropagation algorithm to compute the gradients to the data nodes with respect to the output (**the backward pass**).

---

two approaches: lazy and eager

lazy execution

Tensorflow 1.x default
used in worksheet 4

- Define the computation graph.
- Compile it.
- Iterate backward/forward over the data

Fast. Many possibilities for optimization. Easy to serialise models.
Difficult to debug. Model must remain static during training.

12

There are two ways of doing this. The first is to use **lazy execution**: you define your computation graph, but you don't place any data in it (only nodes that will hold the data later). Then you compile the computation graph and start feeding data through it.

The drawback of this sort of model is that when something goes wrong during the forward pass, it's very difficult to trace the program error (which happens after you compiled the computation graph) back to where you actually made the mistake (somewhere during definition of the computation graph).

two approaches: lazy and eager

eager execution

PyTorch, Tensorflow 2.0 default.
Option for Tensorflow 1.x

- Build the computation graph on the fly during the forward pass.

Easy to debug, problems in the model occur as the module is executing. Flexible: the model can be entirely different for each forward pass.
More difficult to optimize. A little more difficult to serialize.

13

Eager execution does not require this kind to predefining of the knowledge graph. You simply use programming statements to compute the forward pass, for instance multiplying two matrices. The deep learning system then ensure that your matrices are special objects, that keep track of the whole computation, so that when it comes time to do the backward pass, we know how to go back through the computation we performed.

Since eager execution seems to be fast becoming the default approach, we will focus on that, and describe how an eager execution deep learning system works.

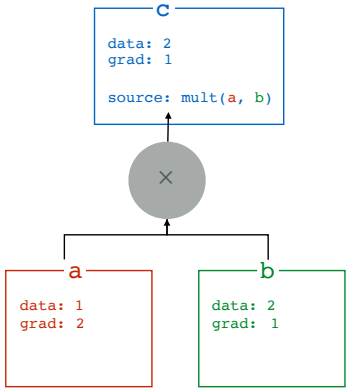
---

example

```
a = Variable(1)
b = Variable(2)

c = a * b

backprop(c)
```



14

In eager mode deep learning systems, we create a node in our computation graph (called a variable here) by specifying what data it should contain. The result is a variable object that stores both the data, and the gradient over that data (which will be filled later).

Here we create the variables a and b. If we now apply an operation (a module) to these, for instance to multiply their values, the result has another variable c. We compute the data stored in c by running the computation, bu we also store references to the variables that were used to create c, and the module that created it.

Using this graph, we can perform the back propagation from a given starting node. We work our way down the graph computing the derivative of each variable with respect to c.

These names and this syntax are loosely inspired by those of PyTorch, but the concepts are similar for all deep learning frameworks.

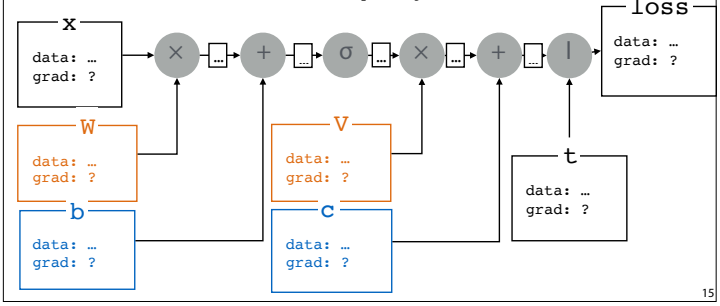
$$\text{loss}(\mathbf{V}\,\sigma(\mathbf{W}\mathbf{x}+\mathbf{b})+\mathbf{c},\mathbf{t})$$

```
W, V, b, c = Variable(…), Variable(…), Variable(…), Variable(…)

for x, t in data:

  h = sigmoid(mult(V, x) + b)
  y = mult(V, h) + c

  loss = l(y, t) # compute the loss
  backprop(loss)
  gradient_descent_step() # perform one step of GD
                          # with the computed gradients
```



15

Here's what a training loop would look like for a simple two-layer feedforward network. The computation graph show below is rebuilt from scratch for every iteration of the training loop, and cleared at the end. The variables W, V, b and c, that define our neural network are re-used and updated every iteration

Note that the output of every module is also a variable, with its own data and its own gradient.

Note that the multiplications are now matrix multiplications

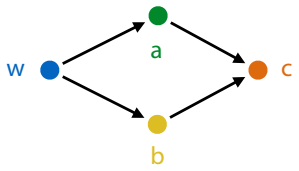
backpropagation revisited

The multivariate chain rule
Dealing with multiple paths in the comp. graph

Backpropagation with tensors
Matrix calculus

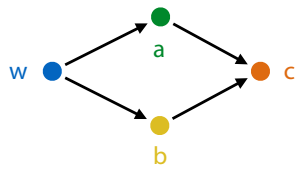
In order to make backpropagation flexible and robust enough to work in this setting, we need to discuss two features that we haven't mentioned yet: how to perform backpropagation if the result depends on the dependent variable along different computation paths, and how to take derivatives when the variables aren't scalars.

diamonds



This is the first problem. If c has two inputs, both depending on w, we can't apply the chain rule. The chain rule we know only works for functions of one input (or one input we're interested in taking the derivative for).

multivariate chain rule



$$\frac{\partial c}{\partial w} = \frac{\partial c}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial c}{\partial b}\frac{\partial b}{\partial w}$$

For such cases, we need the **multivariate chain rule**. It's very simple: to work out the derivative of a function with multiple inputs we just take a single derivative for each input (treating the others as constants), and sum them.

matrix calculus

Can we express the local derivatives in terms of the *tensor* inputs and outputs?

$$l = \sum (\mathbf{y} - \mathbf{t})^2$$
$$\mathbf{y} = \mathbf{V}\mathbf{h} + \mathbf{c}$$
$$\mathbf{h} = \sigma(\mathbf{k})$$
$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

What does it mean to take the derivative *of a vector*, or *with respect to a matrix*?

---

scalar over vector

$$f(\mathbf{a}) = \mathbf{a}^\mathsf{T}\mathbf{b}$$

$$\frac{\partial \mathbf{a}^\mathsf{T}\mathbf{b}}{\partial \mathbf{a}}$$

$$\frac{\partial \mathbf{a}^\mathsf{T}\mathbf{b}}{\partial a_3} = \frac{\partial a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4}{\partial a_3} = b_3$$

$$\frac{\partial \mathbf{a}^\mathsf{T}\mathbf{b}}{\partial \mathbf{a}} = [b_1, b_2, b_3, b_4]^\mathsf{T}$$

(this is just the gradient)

Let's start simple: say we have a function that takes a vector argument, and returns a scalar.

The main trick to working out what a matrix derivative means is always to think the derivative of one element in the output over one element in the input (just pick two random ones). Once you've worked out what the scalar derivatives are, you can usually see how they form a matrix or vector together.

Normally, you need to worry about how exactly to shape such an output (like whether to form a column or a row vector), but in our case, as we'll see, we don;don't usually need to worry about that.

We're using a dot product just to show the principle on a very simple function, but this applies to any function that takes a vector and somehow turns it into a scalar.

---

vector over vector

$$f(\mathbf{a}) = \mathbf{B}\mathbf{a}$$

$$\frac{\partial \mathbf{B}\mathbf{a}}{\partial \mathbf{a}}$$

$$\frac{\partial (\mathbf{B}\mathbf{a})_2}{\partial a_3} = \frac{\partial \mathbf{B}_{2\cdot} \cdot \mathbf{a}}{\partial a_3} = \frac{\partial B_{21} a_1 + B_{22} a_2 + B_{23} a_3}{\partial a_3} = B_{23}$$

$$\frac{\partial \mathbf{B}\mathbf{a}}{\partial \mathbf{a}} = \mathbf{B}$$

Here is another example: a function that takes a vector and produces a vector. We simplify by working out the derivative of the second element of the output over the third element of the input.

function returns a

input is a	scalar	vector	matrix
scalar	scalar	vector	matrix
vector	vector	matrix	?
matrix	matrix	?	?

$$\frac{\partial \mathrm{l}}{\partial \mathbf{W}} = \frac{\partial \mathrm{l}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

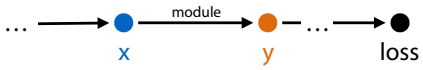
22

The matrix derivative is just an arrangement of every possible scalar derivative: every possible scalar element in the output over every possible scalar element in the input. This tells us what to expect: if we have a vector input and a vector output, all pairs of elements from the two are naturally arranged in a **matrix**. For certain combinations (vector/matrix, matrix/matrix) the results are tensors of order 3 or higher. You can still work these out, but the added complexity doesn't really pay off.

Luckily, we can use a trick to simplify things for ourselves: the *global* derivative that we are ultimately interested is always in the leftmost column: because our computation always results in a scalar loss, we are ultimately only interested in the derivative of a scalar over an n-tensor of weights. The resulting derivative of that has the exact same shape as the weight tensor itself.

The intermediate (local) derivatives may result in complex high-order tensors, but we can avoid those by not computing the local derivatives explicitly, but instead *accumulating them left to right*.

solution: accumulate the gradient product



forward(x): given input x, compute output y.

backward(l_y): given $\mathrm{l}_y = \frac{\partial \mathrm{loss}}{\partial \mathbf{y}}$

compute $\frac{\partial \mathrm{loss}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}$

23

So instead of letting the backward function compute the local derivative explicitly, we give it the derivative of the loss over the output, and ask it to multiply that with the local derivative of the output over the input.

This multiplication can be done without computing the local derivative explicitly. Let's look at an example.

for example

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\frac{\partial \mathrm{l}}{\partial \mathbf{W}} = \frac{\partial \mathrm{l}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{h}} \frac{\partial \mathbf{h}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

forward(**W**, **x**, **b**): compute **W**x + **b**

backward(l_k): compute

$$\frac{\partial \mathrm{l}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}, \quad \frac{\partial \mathrm{l}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{x}}, \quad \frac{\partial \mathrm{l}}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{b}}$$

24

This is the first layer of our feedforward network. It has three inputs **W**, **x**, and **b**, and one output **k**. That means the backward function is given the derivative of the loss over **k**, and should output the product of that derivative with three local derivative: **k** over **W**, **k** over **x** and **k** over **b**.

We'll look at the first as an example. The local derivative **k** over **W** is a 3 tensor which is difficult to work with, and would take up a large amount of memory, so we don't want to compute it explicitly. Let's see if we can work out a more efficient expression for the required product.

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

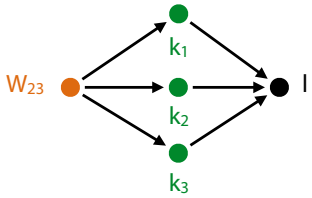
$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial W_{23}}$$

25

As before, we'll first simplify, by looking at the scalar derivative for just one element of W. This doesn't get rid of the vector k, however: both sides of the product still represent multiple derivatives, and we haven't defined how to multiply them.

In this case they're vectors, and we know how to multiply vectors, but we'd like to derive a general approach that also works if **k** is complicated tensor that we don't know how to multiply. We can do this with the multivariate chain rule.

multivariate chain rule



$$\begin{aligned}
&\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial W_{23}} \\
&= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial k_i}{\partial W_{23}} \\
&= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (\mathbf{W}\mathbf{x} + \mathbf{b})_i}{\partial W_{23}} \\
&= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial (W_{i\cdot}{}^{\mathsf{T}} \mathbf{x})}{\partial W_{23}} \\
&= \sum_i \frac{\partial l}{\partial k_i} \frac{\partial \sum_j (W_{ij} x_j)}{\partial W_{23}} \\
&= \sum_{i,j} \frac{\partial l}{\partial k_i} \frac{\partial (W_{ij} x_j)}{\partial W_{23}} = \frac{\partial l}{\partial k_2} x_3
\end{aligned}$$

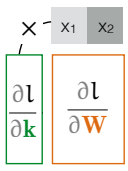
26

If we think of this as a computation graph with scalar nodes, **k** just represents different inputs to the function that ultimately computes l. That means that the derivative of l over W_{23} is just the sum of the derivatives through each element of k. This works whatever the rank and shape of **k**; it could be a huge 9-tensor, and all we have to do is flatten it, and sum over its derivatives.

This is not an efficient way of actually computing the required product, it allows us to work out that the derivative of the loss with respect to element (2, 3) of **W** is the product of element 2 of the incoming vector and element 3 of the input.

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial W_{ij}} = \frac{\partial l}{\partial k_i} x_j$$

$$\frac{\partial l}{\partial \mathbf{k}} \frac{\partial \mathbf{k}}{\partial \mathbf{W}} = \frac{\partial l}{\partial \mathbf{k}} \mathbf{x}^{\mathsf{T}}$$



27

This means that we can compute the derivatives of all elements of the matrix **W** (which is itself a matrix of the same size and shape as **W**) by taking the **outer product** of the given vector l over **k**, and the input **x**.

for example

$$\mathbf{k} = \mathbf{W}\mathbf{x} + \mathbf{b} \qquad \frac{\partial l}{\partial \mathbf{W}} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial h}\frac{\partial h}{\partial k}\frac{\partial \mathbf{k}}{\partial \mathbf{W}}$$

forward($\mathbf{W}$, $\mathbf{x}$, $\mathbf{b}$): compute $\mathbf{W}\mathbf{x} + \mathbf{b}$

backward(l_k): compute

$$\frac{\partial l}{\partial \mathbf{k}}\frac{\partial \mathbf{k}}{\partial \mathbf{W}}, \quad \frac{\partial l}{\partial \mathbf{k}}\frac{\partial \mathbf{k}}{\partial \mathbf{x}}, \quad \frac{\partial l}{\partial \mathbf{k}}\frac{\partial \mathbf{k}}{\partial \mathbf{b}}$$

$$\frac{\partial l}{\partial \mathbf{k}}\mathbf{x}^T, \quad \mathbf{W}^T\frac{\partial l}{\partial \mathbf{k}}, \quad \frac{\partial l}{\partial \mathbf{k}}$$

28

The derivatives over **x** and **b** can be worked out in the same way.

working out the backward function

Usually not necessary, only if you write your own module

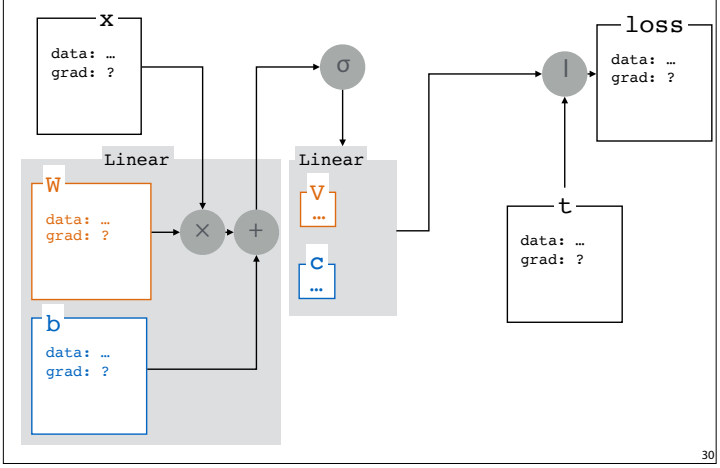
But it's good to understand the principle

- Phrase the problem in terms of scalar derivatives: work out the derivative for one element of the input.
- Use the multivariate chain rule: sum over all elements of the output variable.
- Work out the general solution in terms of matrix operations

Further reading:
https://compsci682-fa18.github.io/docs/vecDerivs.pdf
http://cs231n.stanford.edu/handouts/derivatives.pdf

29

modules / layers



30

Most deep learning frameworks also have a way of combining model parameters and computation into a single unit, often called a **module** or a **layer**.

In this case a Linear module (as it is called in Pytorch) takes care of implementing the computation of a single layer of a neural network (sans activation) and of remembering the weights and the bias.

break

Making it work

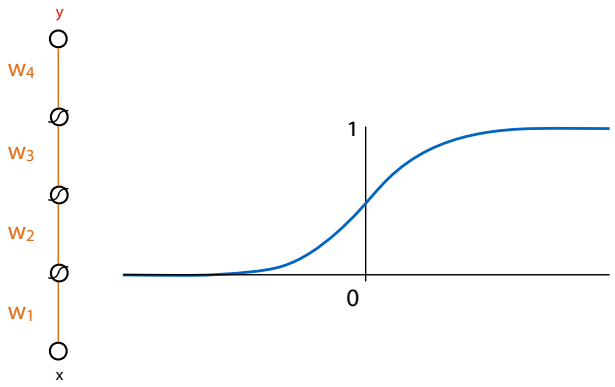
How to make *deep* neural nets work.

- Overcoming vanishing gradients
  Proper initialisation, ReLUs over sigmoids
- Minibatch gradient descent
- Optimizers
  (Nesterov) momentum, Adam
- Regularizers
  L1, L2, Dropout

32

These are the four most important tricks that we use to train neural networks that are big (many parameters) and deep (many layers). Consequently, they are also the main features of any deep learning system.

vanishing gradient problem



33

Here is a simple network to illustrate the problem of **vanishing gradients**. The question is how should we initialize its weights? If we set them too large, the activations will hit the rightmost part of the gradient. Consequently, the local gradient for each node will be very close to zero. That means that the network will never start learning.

If we go the other way, and make the weights large negative numbers, then we hit the leftmost part of the sigmoid and we have the same problem.

initialization

Make sure your data is standardized/normalized

i.e. the mean is close to **0**, and the variance is close to 1 in every direction.

initialise weights W:

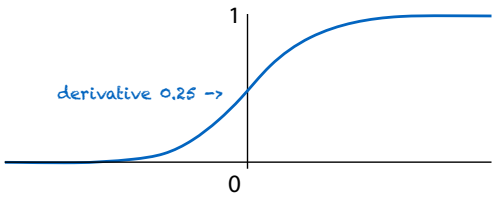
- Make W a random orthogonal matrix (eigenvalues all 1).
- Glorot uniform:

$$w_{ij} \sim U\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

34

There are two standard initialisation mechanisms. The idea of both is that we assume that the layer input is (roughly) distributed so that its mean is 0 and the variance is 1 in every direction (we must standardise of normalise the data so this is true for the first layer).

The initialisation is then designed to pick a random matrix that keeps these properties true (in a stochastic sense).

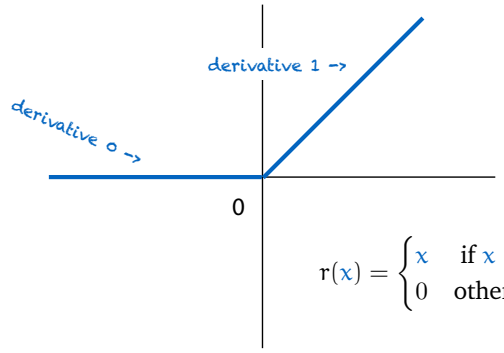


35

But our troubles aren't over. Even if the value going in to the sigmoid is close enough to zero, we still end up with a derivative of only one quarter. This means that propagating the gradient down the network, it will still go to zero with many layers.

We could fix this by squeezing the sigmoid, so its derivative is 1, but it turns out there is a better and faster solution that doesn't have any of these problems.

ReLU



$$r(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

36

The rely activation preserves the derivatives for the nodes whose activations it lets through. It kills it for the nodes that produce a negative value, of course, but so long as your network is properly initialised, about half of the values in your batch will always produce a positive input for the ReLU.

There is still the risk that during training, your network will move to a configuration where a neuron always produces negative input for every instance in your data. If that happens, you end up with a dead neuron: its gradient will always be zero and no weights below that neuron will change anymore (unless the also feed into a non-dead neuron).

minibatch gradient descent

Like stochastic gradient descent, but with small batches of instances, instead of single instances.

Smaller batches: more like stochastic GD, more noisy, less parallelism.

Bigger batches: more like regular GD, more parallelism, limit is (GPU) memory

General advice, keep it between 16 and 128 instances.

optimizers

Attempt to adapt the gradient descent update rule to improve convergence.

- **Momentum**
- **Nesterov momentum**
- RMSProp
- AdaGrad
- AdaDelta
- AdaMax
- **Adam**
- Nadam

General advice, use Adam, try Nesterov momentum if it doesn't work.

momentum

plain gradient descent

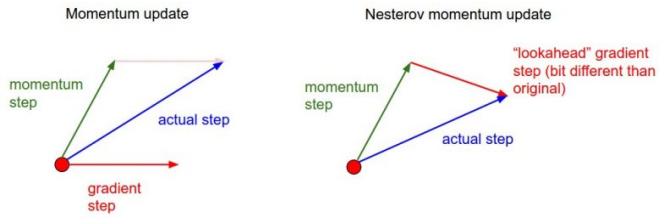
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \text{loss}(\mathbf{w})$$

with momentum

$$\mathbf{v} \leftarrow \mu \mathbf{v} - \eta \nabla \text{loss}(\mathbf{w})$$
$$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{v}$$

If gradient descent is a hiker in a snowstorm, then moment gradient descent is a boulder rolling down a hill. The gradient doesn't affect its movement directly, it acts as a **force** on a moving object. If the gradient is zero, the updates continue in the same direction as the previous iteration, only slowed down by a "friction constant" mu.

Nesterov momentum



source: **http://cs231n.github.io/neural-networks-3/#sgd**
(recommended reading)

Nesterov momentup is a slight tweak. In regular momentum, the actual stem taken is the sum of two vectors, the momentum step (a step in the direction we took last iteration) and a gradient step (a step in the direction of steepest descent at the current point).

Since we know that we are taking the momentum step anyway, we might as well evaluate the gradient *after* the momentum step. This will make the gradient slightly more accurate.

---

adam

Normalize the gradients: keep a running mean **m** and (uncentered) variance **v** for each parameter over the gradient. Subtract these instead of the gradient.

$$\mathbf{m} \leftarrow \beta_1 * \mathbf{m} + (1 - \beta_1)\nabla\text{loss}(\mathbf{w})$$
$$\mathbf{v} \leftarrow \beta_2 * \mathbf{v} + (1 - \beta_2)(\nabla\text{loss}(\mathbf{w}))^2$$
$$\mathbf{w} \leftarrow \mathbf{w} - \eta\frac{\mathbf{m}}{\sqrt{\mathbf{v}} + \epsilon}$$

Comes with three extra hyperparameters, but the defaults are usually fine.

Adam combines the idea of momentum with the idea that in large, complex networks each weight should have its own learning rate. Different weights perform very different functions, so ideally we want to look at the properties of the loss landscape for each weight (the sizes of recent gradients) and scale the "global learning rate" by these.

The bigger the recent gradients, the bigger we want the learning rate to be. However, if there is a lot of variance in the recent gradients, we want to reduce the learning rate (because the landscape is unpredictable). Thus, if we scale the learning rate by the mean **m** over the recent gradients, and divide that by the square root of the variance **v** (plus some small epsilon to avoid division by zero), we end up with a direction that uses recent information about the loss landscape to adapt the gradient to the local topology.

m and **v** are computed as an exponential moving average. This means that the current gradient weights the most, and the influence of recent gradients decays exponentially (but all play *some* part in the total sum).

---

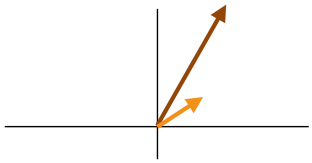
regularisers

The bigger your model, the greater the capacity for **overfitting**.

Regularizers attempt to pull the model back towards more simple models, without eliminating the more complex solutions.

L2 regularizer

$$\text{loss}_{\text{reg}} = \text{loss} + \lambda||\theta||$$



model space

43

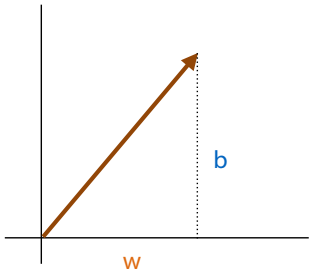
The l2 regularizer considers models with small parameters to be simpler (and therefore preferable). It adds a a penalty to the loss for models with larger weights.

To implement the regularizer we simply compute the l2 norm of all the weights of the model (flattened into a vector). We then add this to the loss multiplied by hyper parameter lambda. Thus, models with bigger weights get a higher loss, but if it's worth it (the original loss goes down enough), they can still beat the simpler models.

Theta is a vector containing all parameters of the model (it's also possible to regularise only a subset).

---

vector norm



$$\theta = \begin{pmatrix} w \\ b \end{pmatrix}$$

$$||\theta|| = \sqrt{w^2 + b^2}$$

$$||\theta||^p = \sqrt[p]{w^p + b^p}$$

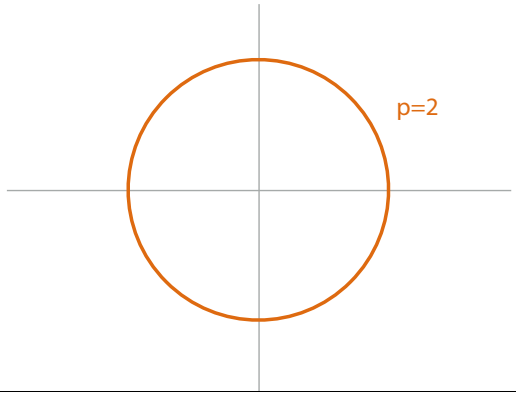
44

We can generalise the l2 norm to an lp norm by replacing the squares with some other number p.

---

lp norm

$$||\theta||^p = \sqrt[p]{w^p + b^p}$$

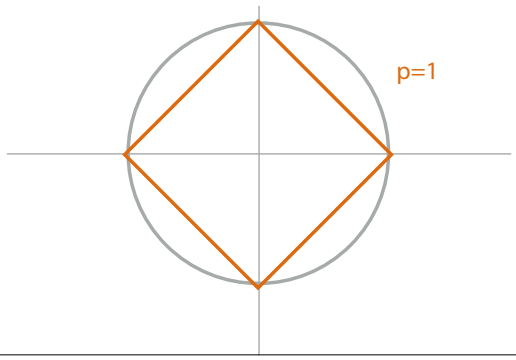


45

For the l2 norm, the set of all points that have distance 1 to the origin form a circle.

lp norm

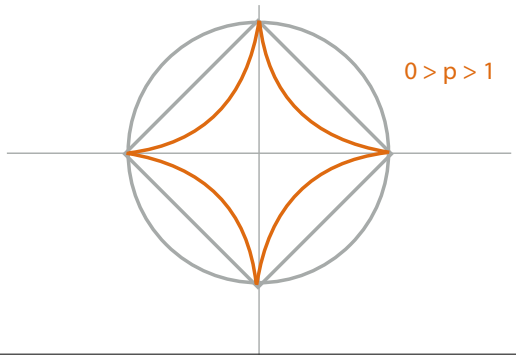
$$||\theta||^p = \sqrt[p]{w^p + b^p}$$



For the l1 norm, the form a diamond.

---

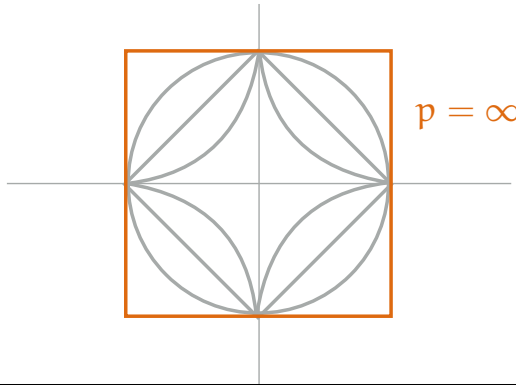
lp norm



---

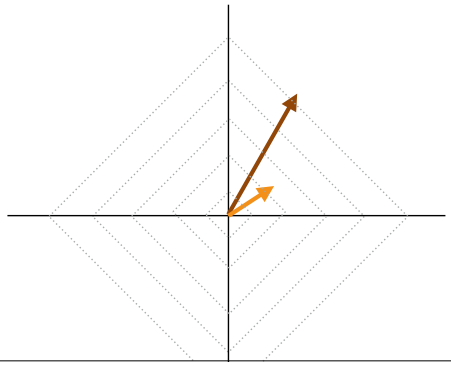
lp norm

$$||\theta||^\infty = \max(w, b)$$



L1 regularizer

$$\text{loss} \leftarrow \text{loss} + \lambda||\theta||^1$$



49

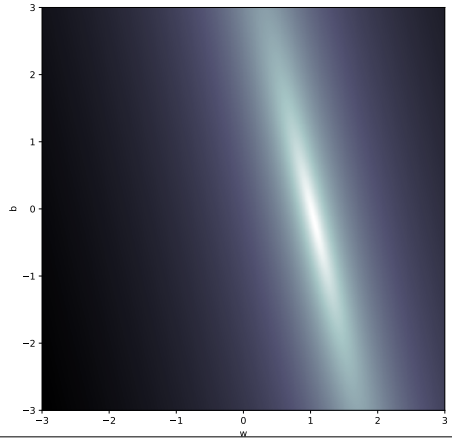
This means that for low p values (p=1 is most common), we get more sparse weight matrices: that is, if a weight is close to 0, it's more likely to get set entirely to zero.

Here's an analogy. Imagine you have a bowl, and you roll a marble down it to find the lowest point. Applying l2 loss is like tipping the bowl slightly to the right. You shift the lowest point in some direction (like to the origin).

51

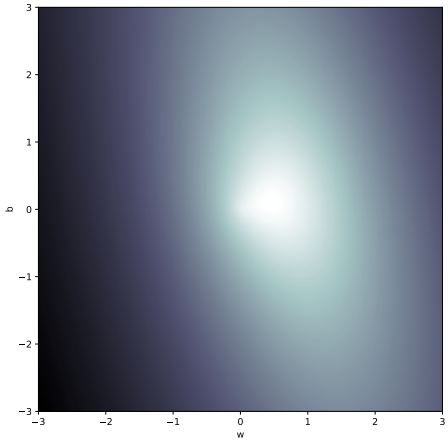
L1 loss is like using a square bowl. It has grooves along the dimensions, so that when you tip the bowl, the marble is likely to end up in one of the grooves.

unregularized



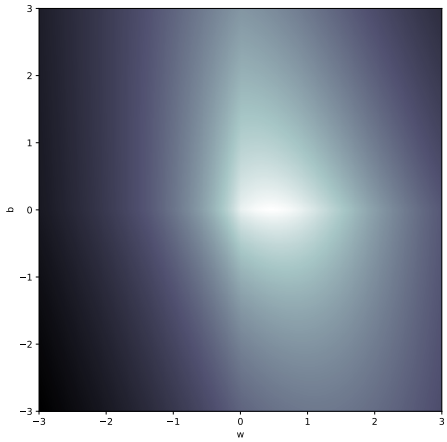
52

L2



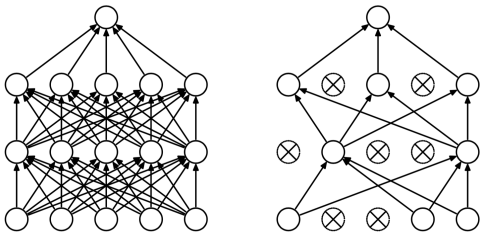
53

L1



54

dropout



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

55

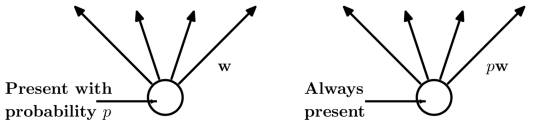
Dropout is another regularization technique for large neural nets. During training, we simply remove hidden and input nodes (each with probability p).

This prevents co-adaptation. Memorization (aka overfitting) often depends on multiple neurons firing together in specific combinations. Dropout prevents this.

image source: http://jmlr.org/papers/v15/srivastava14a.html

---

dropout



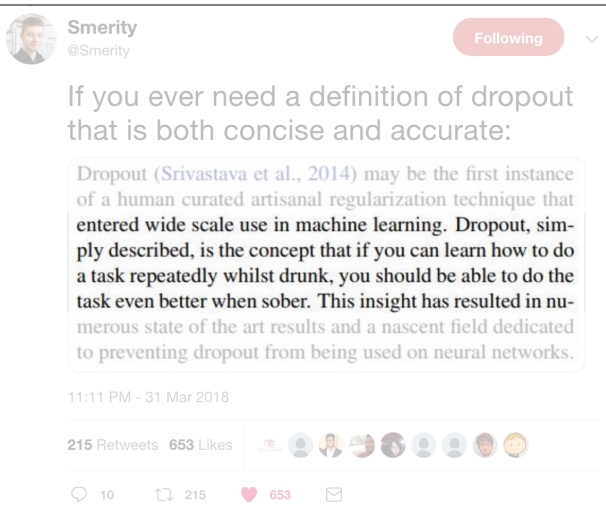
(a) At training time (b) At test time

Figure 2: **Left**: A unit at training time that is present with probability p and is connected to units in the next layer with weights **w**. **Right**: At test time, the unit is always present and the weights are multiplied by p. The output at test time is same as the expected output at training time.

56

Once you've finished training and you starting using the model, you turn off dropout. Since this increases the size of the activations, you should correct by a factor of p.

Frameworks like Keras know when you're using the model to train and when you're using it to predict, and turn dropout on and off automatically.



57

https://twitter.com/Smerity/status/980175898119778304

summary: regularization

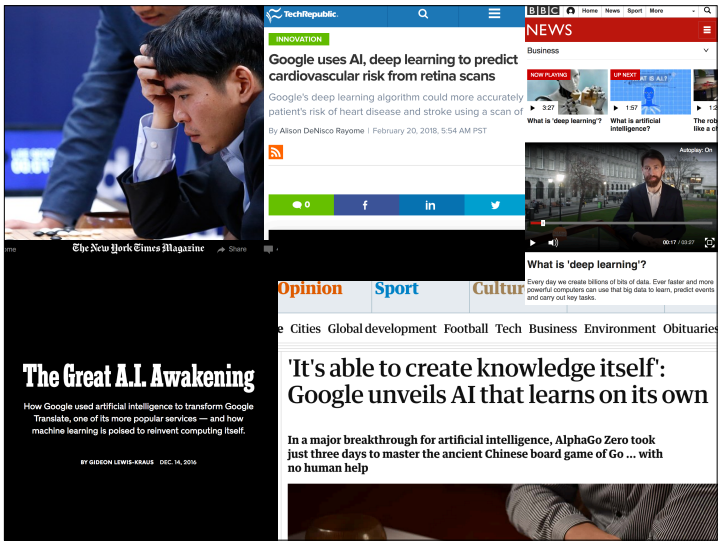
Provides a soft preference for "simpler" models.

L2: Simpler means smaller parameters

L1: Simpler means smaller parameters and more zero parameters

Dropout: randomly disable hidden units. Simpler means more robust

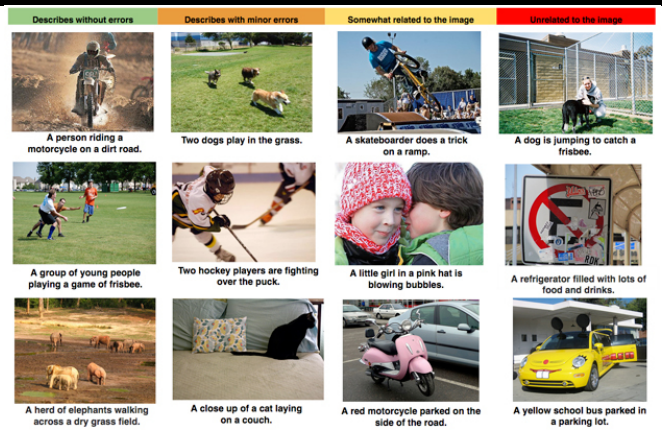
Many other tricks available for regularisation.



Once we had the basic frameworks for deep learning worked out and we started to get the hang of training big and deep networks, the deep learning revolution started to get going. Let's look at some early successes (mostly in the visual domain).

---

natural language descriptions (2014)



This is an end-to-end system for producing natural language descriptions of photographs. The system is not provided with any knowledge of the way language works, it just learns to produce captions from examples using a single neural network that consumes images and produces text, trained end-to-end.

source: ***https://www.engadget.com/2014/11/18/google-natural-language-image-description/***
recommended: ***https://twitter.com/picdescbot***

Style transfer (2015)

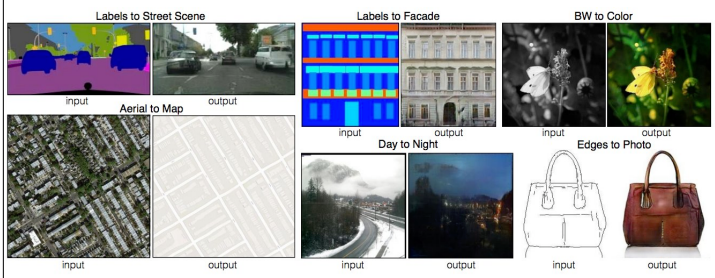
61

This example uses a convolutional network to transfer the style of one image onto another. Interestingly, this work was done with a general purpose network, trained on a general classification task (such networks are available for download). The authors took this network, and didn't change the weights. They just built the style transfer architecture around the existing network.

source: ***https://research.googleblog.com/2016/10/supercharging-style-transfer.html***
try it yourself: ***http://demos.algorithmia.com/deep-style/***

---

image-to-image (2016)



62

This is pix2pix: a network with images as inputs and images as outputs was trained on various example datasets. Note the direction of the transformation. For instance, in the top left, the street scene with labeled objects was the *input*. The car-like objects, road surface, tree etc were all generated by the neural network to fill in the coloured patches in the input. Similarly, the bottom right shows the network generating a picture of a handbag *from a line-drawing*.

---

unmatched image to image (2017)

63

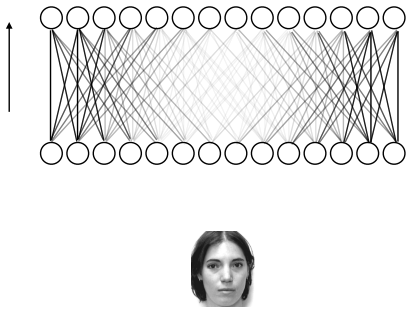
In some cases, we don't have neatly paired images: like the task of transforming a horse into a zebra. We can get a big bag of pictures of horses, and a big bag of pictures of zebras, but we don't know what a specific horse should look like as a zebra. The CycleGAN, published in 2017, could learn in this setting.

http://gntech.ae/news/cyclegan-ai-can-turn-classic-paintings-photos/

https://twitter.com/goodfellow_ian/status/1084973596236144640

---

convolutional layers: pruning and weight sharing

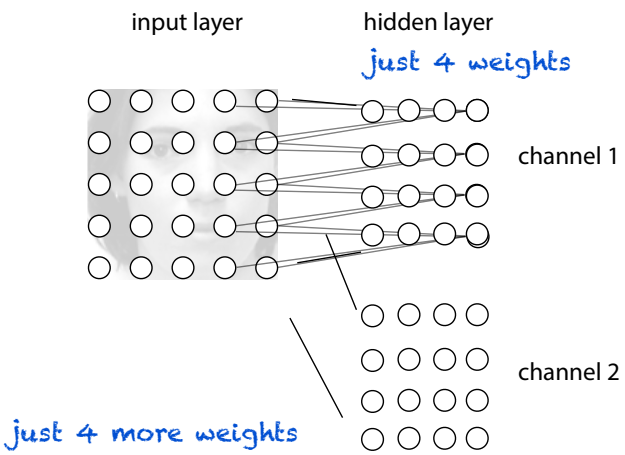


To get started with deep learning, let's look t our first special layer. That is, a layer that is now just a fully connected linear transformation of a vector, but a layer whose shape is determined by some knowledge about its purpose. In the case of the convolution, its purpose is to consume *images*.

We know that images form a grid, and we can use this information to get far fewer connections, and far fewer weights in the layer than a fully connected layer.

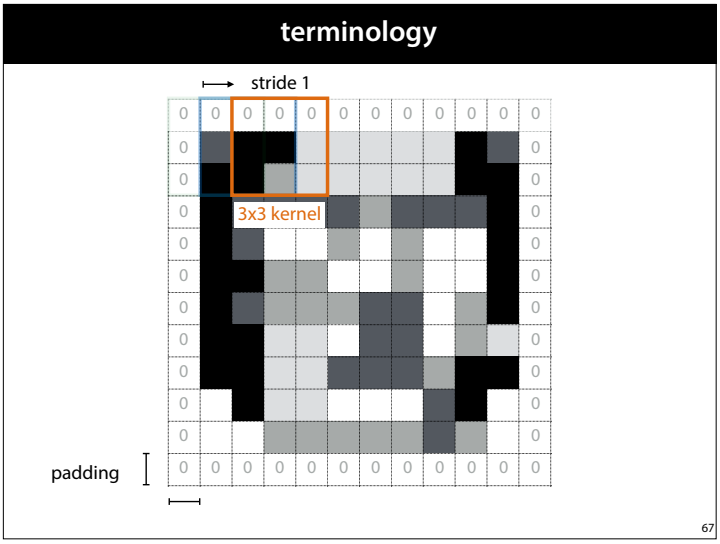
We will start by

Convolutional neural network

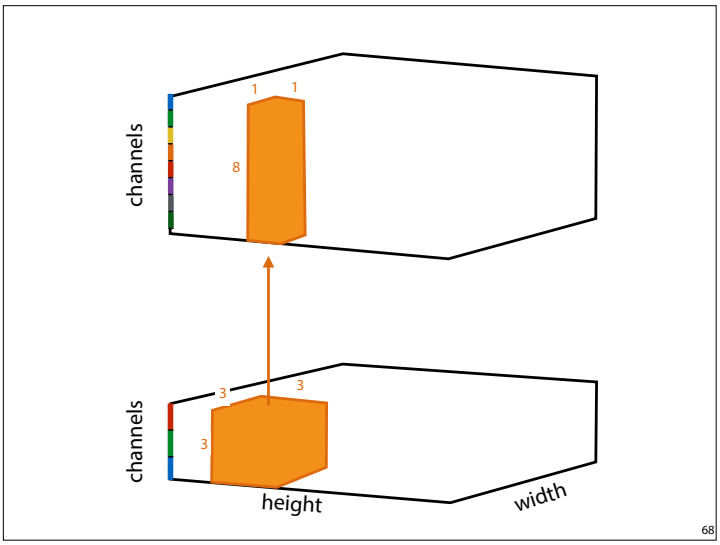


We connect each node in the hidden layer just to a small n by n neighbourhood in the input (here n=2); there are no connections to any other pixels. We do this for each such n x n neighbourhood in the input. For an input image of 5 by 5 pixels, this gives us an input layer or 25 nodes, and a hidden layer of 16 nodes (which we've also arranged in a grid). Each node in the hidden layer has just 4 incoming connections. What's more, we set the 4 weights of these incoming connections to be the same for each of the 16 nodes in the hidden layer.

To extend the hidden layer, we can add additional **channels** to the hidden layer. For an extra channel we follow the same procedure but with 4 new weights. If, as shown here, we have a 5 by 5 input layer with 4 pixel neighbourhoods, and two maps, we get a network with 25 inputs and 32 nodes in the hidden layer. In a traditional feedforward network, that would give us 25 x 32 connections with as many weights. Here, we have just 32 x 4 connections, and only 8 weights.

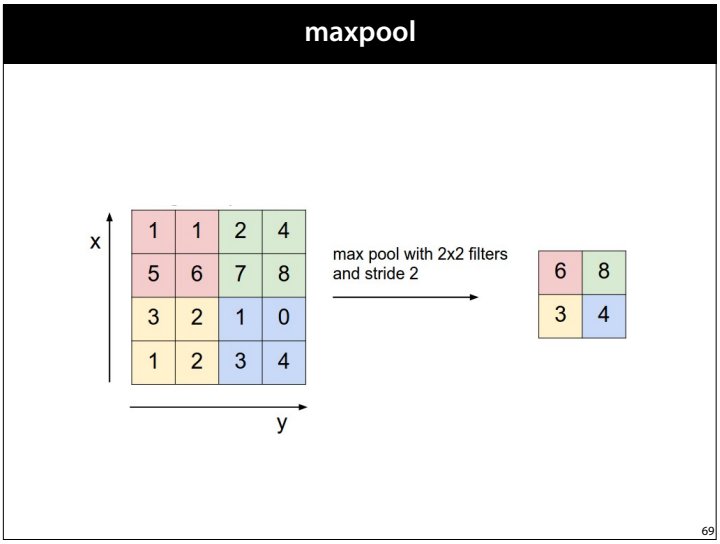


This “sliding window” (called a **kernel**) is usually a square grid with an odd number of pixels per side (i.e. it has a middle pixel). We usually pad the input image with zeros so that the output layer has the same resolution as the input image.



Used in this way, the convolution layer transforms the input, a 3-tensor, into another 3-tensor with the same resolution and more channels.

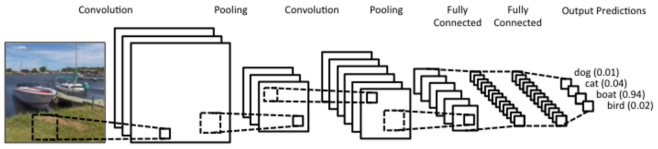
Between the two orange boxes, everything is fully connected.



We chain these convolutions together, but after a while (as the number of channels grows) we’d like the resoltion to decrease so we’re gradually looking at less specific parts of the image, but we have more information (more channels) about that part of the image.

The max pooling layer does this for us, it divides the image into n-by-n squares, and returns the maximum value from each square. Average pooling (returning the average over each square) is also possible, but max pooling is usually more effective.

Most likely, this is because during the backward pass, the whole gradient flows back down one of the pixels, instead of dividing over all four. This gives the backpropagation a sparse character: it focuses only on the most important paths in the computation graph instead of dividing its attention over all of them.



ReLU activations after each layer.

source: **http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/**

70

We extend this principle for many more layers. At each step the maps of the layers get smaller, and we add more maps. Eventually, we add one or two fully connected layers, and a (softmax) output layer (if we're doing image classification).

---

worksheet 4

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Conv2D, MaxPool2D, Dropout, Flatten

model = Sequential()
model.add(Conv2D(16, kernel_size=(3, 3), activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPool2D(pool_size=(2, 2)))
model.add(Dropout(0.25)) # Dropout 25% of the nodes of the previous layer during training
model.add(Flatten())     # Flatten, and add a fully connected layer
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax')) # Last layer: 10 class nodes, with dropout
model.summary()
```

```
from keras.optimizers import Adam

optimizer = Adam()
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model, iterating on the data in batches of 32 samples
model.fit(x_train, y_train, epochs=15, batch_size=32, validation_split=1/6)
```

71

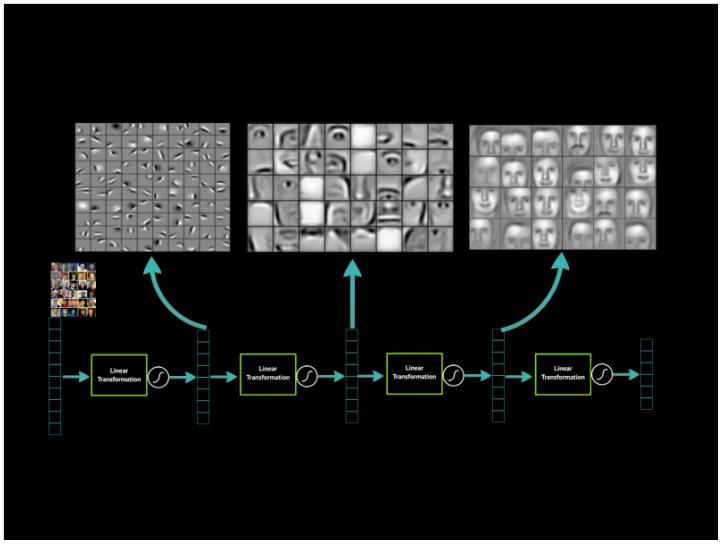
---

filters

72

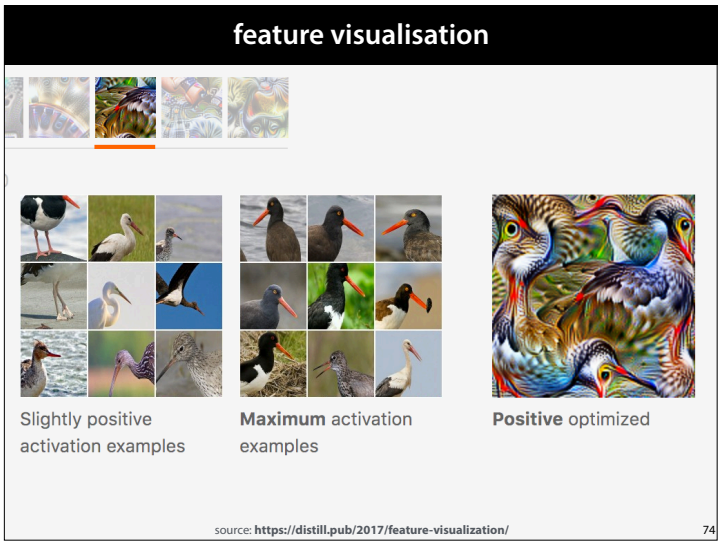
Each first layer gives us several **convolution filters** (one per channel). Here is an example: the Gaussian convolution. It takes a pixel neighbourhood and averages the pixels in it, creating a blurred result of the input (this is just one transformation that a convolution filter can perform, depending on the weights, many other operations are possible).

While Gaussian blur, may seem to be throwing away valuable information, what we actually get is a representation that is *invariant* to noise. All these noisy input images in the left will be mapped to the same image on the right. We can do the same thing to create representations invariant to, for instance, small translation.



Here are the results of a real convolutional network trained to detect faces. The small grayscale images shows a typical image that each node in one of the layers responds to. Those for the first layers can be thought of as edge detectors: if there is a strong edge in a particular part of the image, the node lights up. The second combine these into detectors for parts of images: eyes noses, mouths, etc. The third combine these into detectors for complete faces.

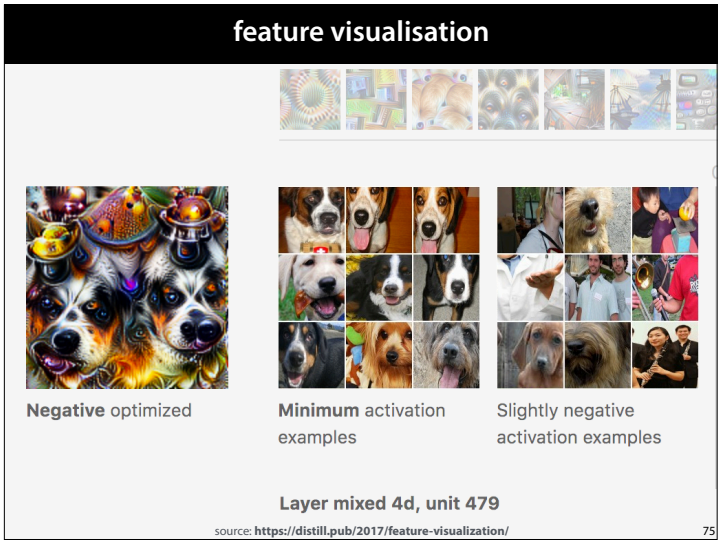
source: ***https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/***



Here is a feature visualisation example for a more recent network trained on imagenet, a collection of 14 million images with diverse subjects.

To find the image on the right, the authors took one node high up in the network, and instead of optimising the weights to minimise the loss, they optimised the input to maximise the activation of that node.

They also searched the dataset for natural images that caused a high activation in that particular node.



The opposite is also possible: searching for an input that cause minimal activation.

https://distill.pub/2017/feature-visualization/

summary: convolutions

Hidden layer has the shape of another image, with more channels.

Hidden nodes are only wired to nearby nodes in the previous layer.

Weights are shared, each hidden nodes has the same weights to the previous layer.

Maxpooling reduces the image dimensions.

end-to-end learning

OCR → tokenization → named entity recognition → relation extraction → "When did Obama become president?"

Finally, let's discuss what deep learning means on a higher level; why we consider it such a departure from classical machine learning.

Here is the kind of pipeline we would often attempt to build in the days before deep learning: we scan old news papers, perform optical character recognition, tokenise the characters into words, attempt to find named entities (like people and companies) and then try to learn the relations between these entities so that we can ask structured queries.

Most of these steps would be solved by some form of machine learning. And after a while, we were getting pretty good at each. So good that it would, for instance, make a mistake for only 1 in a 100 instances.

But chaining together modules that are 99% accurate does not give you a pipeline that is 99% accurate. Error *accumulates*. The tokenization works slightly less well than on its pristine test data, because it's getting noisy input from the OCR. This makes the input for the NER module even more noisy and so on.

end-to-end learning

OCR → tokenization → named entity recognition → relation extraction → "When did Obama become president?"

error signal

The solution of **deep learning** is to make each module differentiable: ensure that we can work out a local gradient so that we can also train the pipeline as a whole using backpropagation.

This is called **end-to-end learning**. other names for roughly the same idea are **differentiable programming** and **programming 2.0**.

what is deep Learning

trad. Machine Learning	Deep Learning
raw data	raw data
feature extraction (hardcoded) ↓	learning ↓
features	hidden layers
learning ↓	learning ↓
output	output
specific learning method for each model	always uses (some form of) gradient descent
classification, regression, clustering	much more flexible (differentiable programming, programming 2.0)

79

Deep learning is to classic machine learning
as Lego is to Playmobil

80

It's a lower level of abstraction, which gives you smaller building blocks. This allows us more creativity in what we can build.
